ETMS - C

E-Commerce and Inventory Management System- C

"A Complete Guide to Understanding the C programme of E-commerce and Inventory management system's backend."

Naim Hossain

April **2026**

About Book

E-commerce and Inventory Management system using C

A comprehensive guide to understanding every line of the Inventory Management System built in C programming language.

Covers: Linked Lists, File I/O, Memory Management, OTP Authentication, Shopping Cart System, Order Processing Pipeline

Version: 2.0 | Year: 2025 | Mirpur, Dhaka, Bangladesh

About Author

Naim Hossain

BSc in Computer Science and Engineering
Daffodil International University

Naim Hossain is a Computer Science and Engineering graduate from Daffodil International University, Dhaka, Bangladesh
He completed his HSC in 2024 from Govt. Science College, Dhaka (GPA 5.00) and his SSC in 2022 from Barura Haji Nowab Ali Pilot High School, Cumilla (GPA 5.00).
Projects: Inventory Management System (C), E-commerce Website, POS Management System, Mini Games
Technical Skills:,JavaScript, C, HTML5, CSS3, React, Node.js, MongoDB, MySQL,, TensorFlow (Basic), Git, GitHub, VS Code, Adobe Photoshop

Leadership: Founder of NextByte IT (2024-present),

Research Interests: Artificial Intelligence, Machine Learning, Data Science, Cybersecurity, Human-Computer Interaction

Teaching Interests: Programming Fundamentals, Data Structures and Algorithms, Object-Oriented Programming, Database Management Systems, Web Development

Contact: hnyaeem264@gmail.com
Location: Mirpur, Dhaka 1216, Bangladesh

Copyright

Copyright © 2025 by Naim Hossain. All rights reserved.

This book and the accompanying source code are the original work of the author. This project was developed as part of Data Structures course work.

Citation: Hossain, N. (2025). Inventory Management System: Complete Code Analysis Guide (Version 2.0). Daffodil International University.

Contact: hnyaeem264@gmail.com

.

Table of Contents

Chapter 1: Introduction to the System 4

Chapter 2: Understanding the Building Blocks (Structures) 5

2.1 The Product Structure 5

2.2 The OrderItem Structure 6

2.3 The Order Structure 6

2.4 The Admin Structure 7

Chapter 3: The Big Picture - How Everything Connects 7

Global Variables (The Heart of the System) 7

Data Flow 8

Chapter 4: Utility Functions - The Helpers 9

4.1 clearScreen() - Cleaning the Console 9

4.2 wait() - Pausing the Program 9

4.3 readString() - Safe String Input 9

4.4 readInt() - Validated Integer Input 9

4.5 readFloat() - Validated Float Input 11

Chapter 5: File Operations - Saving & Loading Data 12

5.1 saveAllData() - Writing to Disk 12

5.2 loadAllData() - Reading from Disk 12

Chapter 6: Admin Authentication System 14

6.1 The Complete Admin Verification Flow 14

6.2 OTP Generation and Sending 15

6.3 Admin Data Storage 15

Chapter 7: Product Management Functions 16

7.1 addProduct() - Adding to Linked List 16

7.2 findProduct() - Searching Linked List 16

7.3 showProducts() - Display All Products 17

7.4 deleteProduct() - Removing from Linked List 18
7.5 updateProduct() - Interactive Product Updates 18
7.6 generateProductId() - Auto-generating IDs 19
Chapter 8: Shopping Cart System 19
8.1 The Cart Structure 19
8.2 addToCart() - Adding Items 20
8.3 showCart() - Display Cart 21
8.4 removeCartItem() - Removing from Cart 21
8.5 clearCart() - Empty Entire Cart 22
Chapter 9: Order Management System 22
9.1 placeOrder() - Creating an Order 22
9.2 findOrder() - Searching Orders 24
9.3 cancelOrder() - Cancelling an Order 24
9.4 updateOrderStatus() - Changing Status 25
9.5 showOrderDetails() - Display Order 25
Chapter 10: Customer Portal - The User Interface 26
10.1 customerMenu() - Main Customer Menu 26
10.2 browseAndBuy() - Shopping Experience 26
10.3 viewCartAndCheckout() - Cart Management 27
10.4 viewMyOrders() - Order History 28
Chapter 11: Admin Portal - The Manager Interface 28
11.1 adminMenu() - Main Admin Menu 28
11.2 adminAddProduct() - Adding with Duplicate Check 29
11.3 adminCreateorder() - Admin Creating Order 29
Chapter 12: The Main Function - Where It All Begins 30
Chapter 13: Complete Flow Diagrams 31
13.1 Complete System Architecture 31
13.2 Data Structure Relationships 31
13.3 Function Call Hierarchy 32

Chapter 14: Common Questions & Answers 33

Appendix: Quick Reference 35

Color Codes Used 35

Common Function Patterns 36

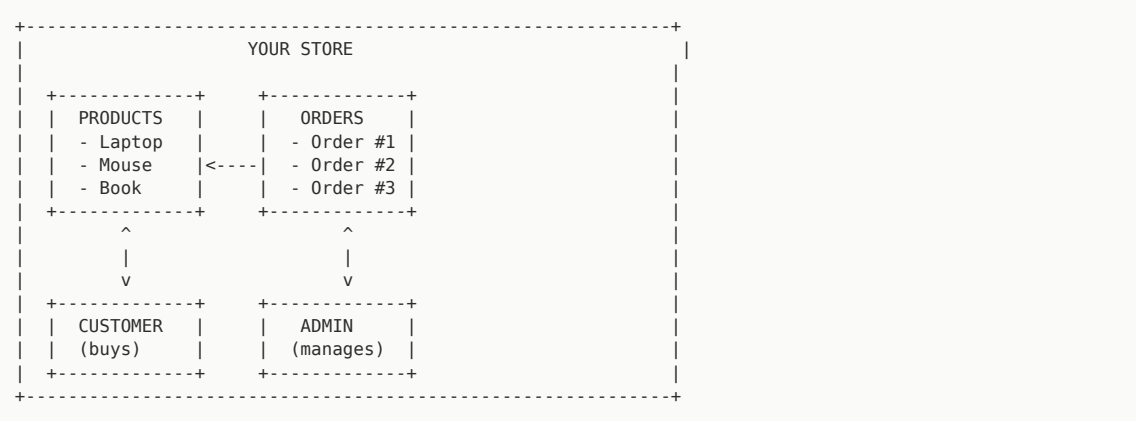
File Format Summary 36

Chapter 1: Introduction to the System

This is an Inventory Management System - a complete software application written in C that helps a store manage its day-to-day operations. The system handles products (what they sell), orders (what customers buy), customers (who buys), and admins (who manages the store). It uses fundamental data structures concepts, particularly linked lists, to organize and manipulate data efficiently.

The system is designed as a console-based application with two separate portals: a Customer Portal for browsing products and placing orders, and an Admin Portal for managing inventory and processing orders. All data persists between sessions through file-based storage, making it a practical demonstration of real-world software engineering principles in C.

Think of it like this:



The system demonstrates several key computer science concepts working together: linked lists for dynamic data storage, file I/O for data persistence, structured programming with modular functions, and memory management using malloc and free. These concepts form the foundation of the Data Structures course and are applied here in a practical, real-world context.

The Four Main Data Types (Structures)

The system uses 4 main structures (like templates for creating data). Each structure serves a specific purpose and connects to the others through pointers, forming a web of relationships that enables the entire system to function:

Structure	Purpose	Real-World Analogy
Product	Stores item information	A product card on Amazon
Order	Stores customer purchase	A receipt / invoice
OrderItem	Items inside an order	Each line on a receipt
Admin	Stores authorized managers	An employee ID card

Table 1.1: The four core data structures and their roles

Chapter 2: Understanding the Building Blocks (Structures)

2.1 The Product Structure

The Product structure is the foundation of the inventory system. Each product in the store is represented as a node in a singly linked list, containing all the information needed to identify and describe the item. The structure includes fields for the product ID, name, quantity in stock, price, category, and a pointer to the next product in the list.

```
typedef struct Product {
    char id[MAX_ID];           // Product ID like "P001"
    char name[MAX_NAME];       // Product name like "Gaming Laptop"
    int quantity;              // How many in stock (50)
    float price;               // Price per item ($999.99)
    char category[MAX_CATEGORY]; // Category like "Electronics"
    struct Product* next;      // Pointer to next product
} Product;
```

What does each part do? The id field provides a unique identifier using the format "P" followed by a zero-padded number (P001, P002, etc.). The name stores the display name. Quantity tracks current stock levels and is modified when orders are placed or cancelled. Price stores the current selling price. Category enables filtering and organization. The next pointer links this product to the next one in the chain, forming the linked list.

Memory Layout:

```
A Product in memory looks like this:

+-------------------------------------------------------------+
|                    Product #1                               |
+-------------------------------------------------------------+
| id:      "P001"                                             |
| name:    "Gaming Laptop"                                    |
| quantity: 50                                               |
| price:   999.99                                            |
| category: "Electronics"                                    |
| next:    -----------------+                                |
+---------------------------+---------------------------------+
                            |
                            v
+-------------------------------------------------------------+
|                    Product #2                               |
+-------------------------------------------------------------+
| id:      "P002"                                             |
| name:    "Wireless Mouse"                                   |
| quantity: 200                                              |
| price:   29.99                                             |
| category: "Electronics"                                    |
| next:    -----------------+                                |
+---------------------------+---------------------------------+
                            |
                            v
                          NULL
```

Why a linked list? Imagine you have products in a store. You do not know how many you will have in advance. An array has fixed size (like a shelf that only holds 100 items), while a linked list can grow and shrink dynamically (like adding more shelves as needed). This is the core advantage of using linked lists over arrays for inventory management.

```
// When you add products, they connect like this:
inventory = p1;  // First product
p1->next = p2;   // p1 points to p2
p2->next = p3;   // p2 points to p3
p3->next = NULL; // Last product points to nothing
```

2.2 The OrderItem Structure

The OrderItem structure represents individual line items within an order. It captures a snapshot of the product information at the time of purchase, which is critically important because product names and prices may change in the future, but historical orders must reflect what the customer actually paid.

```
typedef struct OrderItem {
    char productId[MAX_ID];      // Which product? "P001"
    char productName[MAX_NAME];  // Product name (saved at purchase time)
    int quantity;                // How many bought? (2)
    float price;                 // Price at purchase time ($999.99)
    struct OrderItem* next;      // Next item in order
} OrderItem;
```

Why store productName AND productId? This is a critical design decision. When a customer buys a product, both the product name and price are saved in the order at the moment of purchase. Even if the product name or price changes later, the order will always show exactly what the customer ordered and what they paid. This is the same principle used by real e-commerce systems like Amazon.

```
ORDER RECEIPT (Saved forever):
+----------------------------------------+
| Gaming Laptop x 2 = $1,999.98          |
| Wireless Mouse x 1 = $29.99            |
+----------------------------------------+

PRODUCT CATALOG (Can change later):
+----------------------------------------+
| Gaming Laptop -> Now called "Gamer Pro"  |
| Wireless Mouse -> Price now $39.99       |
+----------------------------------------+

The receipt still shows original names and prices!
```

2.3 The Order Structure

The Order structure ties everything together. Each order contains customer information, the total amount, current status, the order date, a linked list of order items, and a pointer to the next order in the global orders list. This structure allows the system to track the complete lifecycle of every transaction from placement through delivery or cancellation.

```
typedef struct Order {
    int id;                         // Order ID: 1000, 1001, 1002...
    char customerName[MAX_NAME];    // "John Doe"
    char phone[MAX_PHONE];          // "1234567890"
    char address[MAX_ADDRESS];      // "123 Main St"
    float total;                    // Total amount: $2,029.97
    char status[20];                // "Pending", "Shipped", "Delivered"
    time_t orderDate;               // When order was placed
    struct OrderItem* items;        // Linked list of items in this order
    struct Order* next;             // Next order in history
} Order;
```

Visual Representation:

2.4 The Admin Structure

The Admin structure is the simplest of the four, containing just an outlet ID and phone number. It is used for OTP-based authentication. Each admin in the system is verified by their outlet ID, and a one-time password is sent to their registered phone number. This two-factor authentication pattern ensures that only authorized personnel can access the admin portal.

```
typedef struct Admin {
    char outletId[MAX_OUTLET_ID];  // Store ID like "STORE001"
    char phone[MAX_PHONE_ADMIN];   // Phone for OTP
    struct Admin* next;            // Next admin
} Admin;
```

How Admin Verification Works:

```
Step 1: Admin enters Outlet ID: "STORE001"
Step 2: System looks up phone number linked to that ID
Step 3: System generates random 6-digit OTP: 123456
Step 4: System "sends" OTP (displays on screen for simulation)
Step 5: Admin enters OTP: 123456
Step 6: If matches -> ACCESS GRANTED!
```

Chapter 3: The Big Picture - How Everything Connects

Global Variables (The Heart of the System)

Global variables are the central nervous system of the program. They persist throughout the entire lifetime of the application and are accessible from any function. The inventory pointer tracks all products, allOrders tracks all orders, currentCart holds the shopping cart, and the counter variables ensure unique IDs for new products and orders.

```
Product* inventory = NULL;      // Head of products linked list
Order* allOrders = NULL;        // Head of orders linked list
OrderItem* currentCart = NULL;  // Head of shopping cart
```

```
int nextOrderId = 1000;         // Next available order number
int nextProductNumber = 1;      // Next product number for ID
```

Memory Layout Visualization:

```
GLOBAL VARIABLES (exist as long as program runs)
================================================

inventory ----------+
                    v
         +---------------------+     +---------------------+     +---------------------+
         | Product: P001       |---->| Product: P002       |---->| Product: P003       |
         | "Laptop"            |     | "Mouse"             |     | "Book"              |
         | qty:50              |     | qty:200             |     | qty:100             |
         +---------------------+     +---------------------+     +---------------------+

allOrders ----------+
                    v
         +---------------------+     +---------------------+
         | Order: #1000        |---->| Order: #1001        |
         | "John Doe"          |     | "Jane Smith"        |
         | Total: $2059.97     |     | Total: $59.98       |
         | Status: "Pending"   |     | Status: "Delivered" |
         +---------------------+     +---------------------+

currentCart -------+
                   v
         +---------------------+
         | OrderItem: P001     |
         | "Laptop" x 1        |
         +---------------------+

nextOrderId = 1002
nextProductNumber = 4
```

Data Flow

Understanding data flow is crucial. Data moves between disk storage (files) and runtime memory (RAM). When the program starts, loadAllData() reads files into linked lists in memory. Throughout execution, functions modify these in-memory structures. When data changes, saveAllData() writes the current state back to disk. This read-modify-write cycle ensures data persistence across sessions.

```
+-------------------------------------------------------------+
|                    DATA FLOW DIAGRAM                        |
+-------------------------------------------------------------+
                    +-------------+
                    |   DISK      |
                    | (Files)     |
                    +------+------+
                           |
              saveAllData() | loadAllData()
                           |
                           v
                    +-------------+
                    |   MEMORY    |
                    | (RAM)       |
                    +------+------+
                           |
        +------------------+------------------+
        |                  |                  |
        v                  v                  v
+---------------+  +---------------+  +---------------+
|  addProduct() |  | placeOrder()  |  | addToCart()   |
|  updateProduct|  | cancelOrder() |  | removeFromCart|
|  deleteProduct|  | updateStatus()|  | showCart()    |
+---------------+  +---------------+  +---------------+
        |                  |                  |
        +------------------+------------------+
                           |
                           v
              Changes saved to DISK
```

Chapter 4: Utility Functions - The Helpers

4.1 clearScreen() - Cleaning the Console

The clearScreen function provides a clean slate for the user interface. It uses a cross-platform approach through the C preprocessor: on Windows systems it executes the "cls" command, while on Linux and macOS it runs "clear". This ensures the program works consistently across different operating systems, making the terminal interface feel like a real application by removing previous output before displaying new screens.

```
void clearScreen(void) {
    system(CLEAR);  // CLEAR is either "cls" (Windows) or "clear" (Linux/Mac)
}
```

4.2 wait() - Pausing the Program

The wait function is essential for user experience. After displaying messages or results, the program pauses execution by calling getchar(), which blocks until the user presses Enter. Without this function, information would flash on screen and disappear before the user could read it. This simple pattern is used throughout the system after every significant action or message display.

```
void wait() {
    printf("\nPress Enter to continue...");
    getchar();  // Waits for user to press Enter
}
```

4.3 readString() - Safe String Input

The readString function is a safe replacement for the dangerous scanf function when reading strings. While scanf("%s") stops reading at the first space (turning "John Doe" into just "John"), fgets reads the entire line including spaces. The function also handles the trailing newline character that fgets inevitably captures by using strcspn to find and replace it with a null terminator.

```
void readString(char* prompt, char* output, int size) {
    printf("%s", prompt);           // Show prompt
    fgets(output, size, stdin);     // Read input safely
    output[strcspn(output, "\n")] = 0;  // Remove the newline
}
```

Step-by-step example:

```
char name[50];
readString("Enter name: ", name, 50);
// 1. Prints: "Enter name: "
// 2. User types: "John Doe\n"
// 3. fgets stores: "John Doe\n\0"
// 4. strcspn finds position of '\n' (position 8)
// 5. Replaces '\n' with '\0': "John Doe\0"
```

4.4 readInt() - Validated Integer Input

The readInt function is a robust input validation wrapper that eliminates the most common source of crashes in C programs: invalid numeric input. It reads input as a string first, then validates that every character is a digit, converts to an integer, and checks that the value falls within the specified range. If any validation fails,

it displays an error and loops until valid input is received. This defensive programming pattern makes the system virtually crash-proof from user input errors.

```
int readInt(char* prompt, int min, int max) {
    char input[100];
    int value, valid;
    do {
        valid = 1;
        printf("%s", prompt);
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = 0;
        if (strlen(input) == 0) valid = 0;
        for (int i = 0; i < strlen(input) && valid; i++) {
            if (!isdigit(input[i])) valid = 0;
        }
        if (!valid) {
            printf("ERROR: Invalid input! Enter a valid number.\n");
        } else {
            value = atoi(input);
            if (value < min || value > max) {
                printf("ERROR: Enter between %d and %d!\n", min, max);
                valid = 0;
            }
        }
    } while (!valid);
    return value;
}
```

Flow Diagram:

```
+----------------------------------------+
| Show prompt: "Enter choice: "          |
| Get user input                         |
+----------------------------------------+
              |
              v
+----------------------------------------+
| Is input empty?                        |
| YES -> valid=0, show error             |
| NO  -> continue                        |
+----------------------------------------+
              |
              v
+----------------------------------------+
| Are all characters digits?             |
| "123" -> YES                           |
| "12a" -> NO -> valid=0, show error     |
+----------------------------------------+
              |
              v
+----------------------------------------+
| Convert to number (atoi)               |
| Check if between min and max           |
| 5 between 1 and 10? YES                |
| 15 between 1 and 10? NO -> error       |
+----------------------------------------+
              |
              v
+----------------------------------------+
| valid = 1?                             |
| YES -> return number                   |
| NO  -> loop back to start              |
+----------------------------------------+
```

Example executions:

```
// User types: "abc"
readInt("Choice: ", 1, 3);
// Output: "ERROR: Invalid input! Please enter a valid number."
// Asks again

// User types: "5"
```

```
readInt("Choice: ", 1, 3);
// Output: "ERROR: Please enter a number between 1 and 3!"
// Asks again

// User types: "2"
readInt("Choice: ", 1, 3);
// Returns: 2
```

4.5 readFloat() - Validated Float Input

The readFloat function extends the validation pattern to floating-point numbers. It validates that the input contains only digits and at most one decimal point, checks that the decimal point is not at the end (like "10."), and ensures the value is non-negative. This function is primarily used for reading product prices, where negative values would be meaningless and malformed decimals would cause data corruption.

```
float readFloat(char* prompt) {
    char input[100];
    float value;
    int valid, decimalCount;
    do {
        valid = 1; decimalCount = 0;
        printf("%s", prompt);
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = 0;
        if (strlen(input) == 0) valid = 0;
        for (int i = 0; i < strlen(input) && valid; i++) {
            if (input[i] == '.') {
                decimalCount++;
                if (decimalCount > 1) valid = 0;
            } else if (!isdigit(input[i])) {
                valid = 0;
            }
        }
        if (valid && decimalCount == 1) {
            char* dotPos = strchr(input, '.');
            if (*(dotPos + 1) == '\0') valid = 0;
        }
        if (!valid) printf("ERROR: Enter a valid number.\n");
        else {
            value = atof(input);
            if (value < 0) { printf("ERROR: Cannot be negative!\n"); valid = 0; }
        }
    } while (!valid);
    return value;
}
```

Validation Examples:

Input	Valid?	Reason
"10"	YES	Simple integer
"10.99"	YES	Valid decimal
".99"	YES	Leading decimal
"10."	NO	Decimal at end
"10.99.00"	NO	Multiple decimals
"abc"	NO	Letters not allowed
"-10"	NO	Negative not allowed

Table 4.1: readFloat() validation examples

Chapter 5: File Operations - Saving & Loading Data

5.1 saveAllData() - Writing to Disk

The saveAllData function is responsible for persisting the entire application state to disk. It uses a pipe-delimited text format with section markers (#PRODUCTS, #ORDERS, #END) to organize different data types within a single file. Each product is written as a line with five fields separated by pipe characters, and each order is written with seven fields. This simple but effective format makes data both human-readable and easy to parse when loading.

```
void saveAllData() {
    saveAdminData();  // Save admin list first
    FILE* f = fopen("store_data.txt", "w");
    if (!f) { printf("Error saving data!\n"); return; }
    fprintf(f, "#PRODUCTS\n");
    Product* p = inventory;
    while (p != NULL) {
        fprintf(f, "%s|%s|%s|%d|%.2f\n",
                p->id, p->name, p->category, p->quantity, p->price);
        p = p->next;
    }
    fprintf(f, "#END\n\n");
    fprintf(f, "#ORDERS\n");
    Order* o = allOrders;
    while (o != NULL) {
        fprintf(f, "%d|%s|%s|%s|%.2f|%s|%ld\n",
                o->id, o->customerName, o->phone,
                o->address, o->total, o->status, (long)o->orderDate);
        o = o->next;
    }
    fprintf(f, "#END\n");
    fclose(f);
}
```

What the saved file looks like:

```
store_data.txt
==========================================

#PRODUCTS
P001|Gaming Laptop|Electronics|50|999.99
P002|Wireless Mouse|Electronics|200|29.99
P003|Python Programming Book|Books|100|39.99
#END

#ORDERS
1000|John Doe|1234567890|123 Main St|2059.97|Pending|1729876543
1001|Jane Smith|9876543210|456 Oak Ave|59.98|Delivered|1729876600
#END
```

How the pipe (|) format works: The pipe character separates fields on each line, making it easy to read back using the strtok function. strtok(line, "|") gets the first field ("P001"), strtok(NULL, "|") gets the second ("Gaming Laptop"), and so on until all fields are extracted. This delimiter-based approach is simple, reliable, and avoids issues with commas or spaces that might appear in product names.

5.2 loadAllData() - Reading from Disk

The loadAllData function reverses the save process. It opens the data file, reads line by line, and uses section markers to determine what type of data is being read. When it encounters #PRODUCTS, it switches to product parsing mode, creating new Product nodes with malloc and populating them from the pipe-delimited fields. The same process applies for #ORDERS. If the file does not exist, it creates a set of five sample products so the system is usable immediately on first run.

```
void loadAllData() {
    loadAdminData();
    FILE* f = fopen("store_data.txt", "r");
    if (!f) { /* Create sample products */ return; }
    char line[500];
    int section = 0;  // 0=unknown, 1=products, 2=orders
    while (fgets(line, sizeof(line), f)) {
        line[strcspn(line, "\n")] = 0;
        if (strcmp(line, "#PRODUCTS") == 0) { section = 1; continue; }
        else if (strcmp(line, "#ORDERS") == 0) { section = 2; continue; }
        else if (strcmp(line, "#END") == 0) { section = 0; continue; }
        if (strlen(line) == 0) continue;
        if (section == 1) {
            Product* p = malloc(sizeof(Product));
            char* token = strtok(line, "|");
            if (token) strcpy(p->id, token);
            token = strtok(NULL, "|");
            if (token) strcpy(p->name, token);
            // ... parse remaining fields and add to linked list
        }
        else if (section == 2) {
            Order* o = malloc(sizeof(Order));
            // ... parse order fields similarly
        }
    }
    fclose(f);
}
```

Step-by-step loading process:

```
START LOAD
    |
    v
Open file "store_data.txt"
    |
    +--- File doesn't exist? --> Create sample products
    |
    v
Read line by line
    |
    v
+---------------------------------------------+
| Line 1: "#PRODUCTS"                         |
| -> section = 1 (now reading products)       |
+---------------------------------------------+
    |
    v
+---------------------------------------------+
| Line 2: "P001|Gaming Laptop|Electronics|50|999.99" |
| -> section = 1, so parse as product         |
| -> strtok gets: "P001", "Gaming Laptop", etc.|
| -> Create Product, add to linked list       |
+---------------------------------------------+
    |
    v
+---------------------------------------------+
| Line 3: "P002|Wireless Mouse|Electronics|200|29.99" |
| -> Parse as product, add to list            |
+---------------------------------------------+
    |
    v
+---------------------------------------------+
| Line 4: "#END"                              |
| -> section = 0 (stop reading products)      |
```

```
+-------------------------------------------+
    |
    v
+-------------------------------------------+
| Line 5: "#ORDERS"                         |
| -> section = 2 (now reading orders)       |
+-------------------------------------------+
    |
    v
Continue until end of file...
    |
    v
Close file
```

Chapter 6: Admin Authentication System

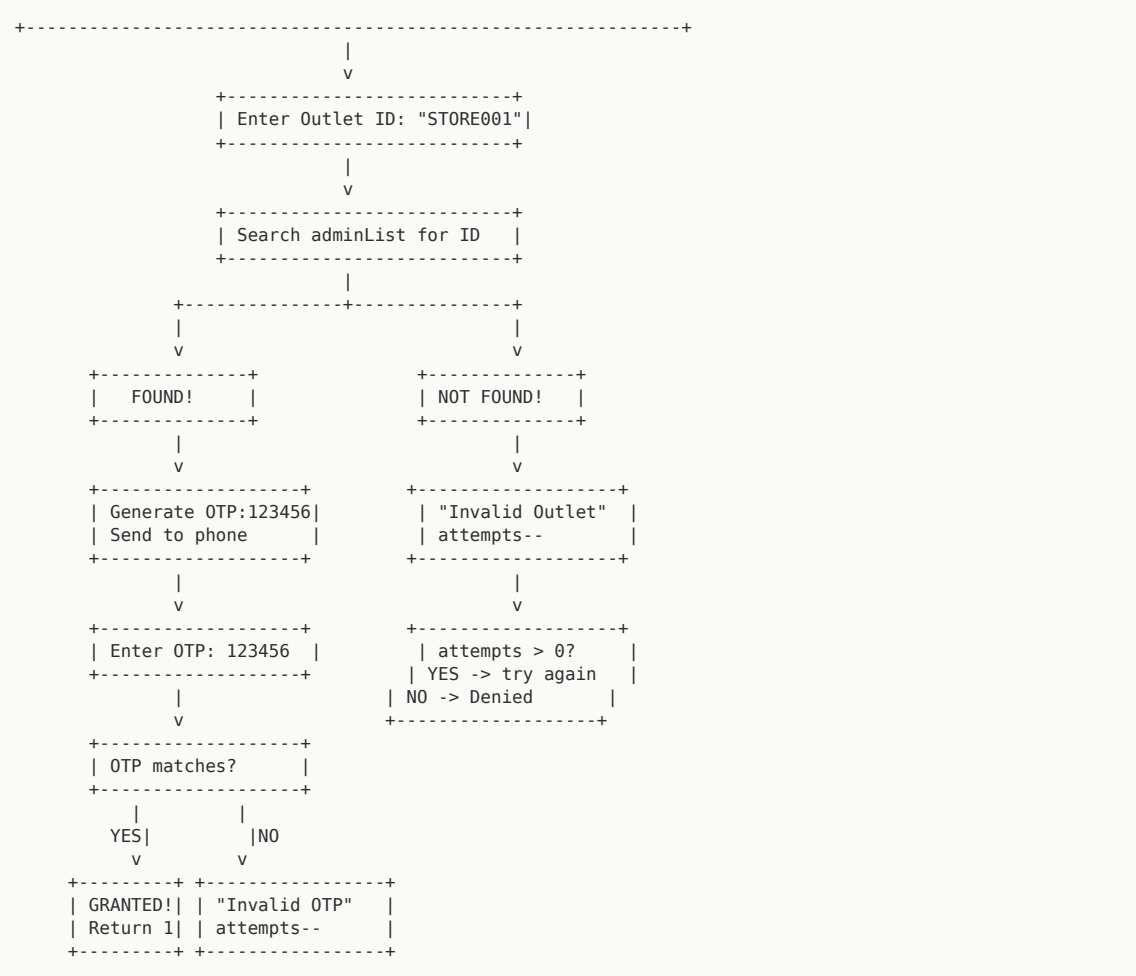
6.1 The Complete Admin Verification Flow

The verifyAdmin function implements a simulated two-factor authentication system. The admin enters their outlet ID, the system searches the admin linked list for a match, generates a random 6-digit OTP, displays it on screen (simulating an SMS send), and then prompts the admin to enter the OTP. The admin gets three attempts before being locked out. This demonstrates the concept of authentication in a tangible way while keeping the implementation simple enough for a student project.

```
int verifyAdmin() {
    char outletId[MAX_OUTLET_ID];
    char inputOTP[7];
    int generatedOTP;
    int attempts = 3;
    while (attempts > 0) {
        printf("\nAttempts left: %d\n", attempts);
        printf("Enter Outlet ID: ");
        fgets(outletId, MAX_OUTLET_ID, stdin);
        outletId[strcspn(outletId, "\n")] = 0;
        Admin* curr = adminList;
        int found = 0;
        while (curr != NULL) {
            if (strcmp(curr->outletId, outletId) == 0) {
                found = 1;
                generatedOTP = generateOTP();
                sendOTP(curr->phone, generatedOTP);
                printf("\nEnter OTP: ");
                fgets(inputOTP, 7, stdin);
                inputOTP[strcspn(inputOTP, "\n")] = 0;
                if (atoi(inputOTP) == generatedOTP) {
                    printf("\n  VERIFICATION SUCCESSFUL!\n");
                    return 1;
                } else {
                    printf("\n  INVALID OTP!\n");
                    attempts--;
                    break;
                }
            }
            curr = curr->next;
        }
        if (!found) { printf("\n  INVALID OUTLET ID!\n"); attempts--; }
    }
    printf("\n  VERIFICATION FAILED!\n");
    return 0;
}
```

Visual Flow of Admin Verification:

```
+-------------------------------------------------------------+
|                 ADMIN LOGIN ATTEMPT                         |
```



6.2 OTP Generation and Sending

The OTP system uses the standard C random number generator seeded with the current time. The srand(time(NULL)) call ensures different random sequences each time the program runs, while rand() % 900000 generates numbers in the range 0-899999. The sendOTP function simulates sending an SMS by displaying the OTP on screen, which is sufficient for a student project demonstration.

```
int generateOTP() {
    srand(time(NULL));           // Seed with current time
    return (rand() % 900000);    // 0 to 899999
}

void sendOTP(char* phone, int otp) {
    printf("\n  OTP SENT TO: %s successful!\n", phone);
    printf("  YOUR OTP IS: %d\n", otp);
    printf("  OTP will expire in 5 minutes\n");
}
```

> Note: This is a simulation! In a real system, the OTP would be sent via an SMS gateway API like Twilio or MessageBird. The security would also include rate limiting, OTP expiration, and encryption.

6.3 Admin Data Storage

Admin credentials are stored separately from product and order data in a dedicated file called admin_data.txt. Each line contains an outlet ID and phone number separated by a pipe character. This separation of concerns

means admin data can be managed independently of store data.

```
admin_data.txt
========================
STORE001|9876543210
STORE002|8765432109
STORE003|7654321098
```

Chapter 7: Product Management Functions

7.1 addProduct() - Adding to Linked List

The addProduct function handles inserting a new product node into the inventory linked list. It follows the standard append-to-end pattern: if the list is empty, the new product becomes the head; otherwise, it traverses to the last node and links the new product there. This ensures products appear in the order they were added, which is intuitive for both display and file saving.

```
void addProduct(Product* p) {
    if (inventory == NULL) {
        inventory = p;  // First product - becomes head
    } else {
        Product* curr = inventory;
        while (curr->next != NULL) {  // Traverse to end
            curr = curr->next;
        }
        curr->next = p;  // Add at the end
    }
}
```

Visual explanation:

```
Case 1: inventory is NULL (empty list)

inventory ----> NULL
                   |
                   v addProduct(p1)

inventory ----> p1 ----> NULL

============================================

Case 2: inventory has products

inventory ----> p1 ----> p2 ----> NULL
                             |
                             v addProduct(p3)

inventory ----> p1 ----> p2 ----> p3 ----> NULL
```

7.2 findProduct() - Searching Linked List

The findProduct function performs a linear search through the inventory linked list. Starting from the head, it compares each product ID with the target using strcmp. When a match is found, it returns a pointer to that product node. If the entire list is traversed without a match, it returns NULL. This is the most common linked list operation and is used extensively throughout the system for validating product IDs, updating stock, and building orders.

```
Product* findProduct(char* id) {
    Product* curr = inventory;
```

```
    while (curr != NULL) {
        if (strcmp(curr->id, id) == 0)  // Compare strings
            return curr;                // Found!
        curr = curr->next;              // Move to next
    }
    return NULL;  // Not found
}
```

Step-by-step search for "P003":

```
findProduct("P003")
    |
    v
curr = inventory (points to P001)
    |
    v
P001.id = "P001" -> compare with "P003" -> NOT equal
    |
    v
curr = curr->next (now points to P002)
    |
    v
P002.id = "P002" -> compare with "P003" -> NOT equal
    |
    v
curr = curr->next (now points to P003)
    |
    v
P003.id = "P003" -> compare with "P003" -> EQUAL!
    |
    v
RETURN pointer to P003
```

7.3 showProducts() - Display All Products

The showProducts function renders the product catalog as a formatted table in the console. It uses printf format specifiers to align columns perfectly, creating a professional-looking display. The function first checks if the inventory is empty, then iterates through the linked list printing each product in a formatted row.

```
void showProducts() {
    if (inventory == NULL) {
        printf("\nNo products available!\n");
        return;
    }
    printf("\n+------------------------------------------------------------------------+\n");
    printf("| %-8s | %-22s | %6s | %8s | %-12s |\n",
           "ID", "Name", "Stock", "Price", "Category");
    printf("+------------------------------------------------------------------------+\n");
    Product* curr = inventory;
    while (curr != NULL) {
        printf("| %-8s | %-22s | %6d | %8.2f | %-12s |\n",
               curr->id, curr->name, curr->quantity,
               curr->price, curr->category);
        curr = curr->next;
    }
    printf("+------------------------------------------------------------------------+\n");
}
```

Format specifiers explained:

Format	Meaning	Example Output
%-8s	String, left-aligned, 8 chars wide	"P001 "
%-22s	String, left-aligned, 22 chars wide	"Gaming Laptop "
%6d	Integer, right-aligned, 6 chars wide	" 50"

Format	Meaning	Example Output
%8.2f	Float, 8 total width, 2 decimals	" 999.99"

Table 7.1: Printf format specifiers used in showProducts()

7.4 deleteProduct() - Removing from Linked List

Deleting a node from a singly linked list requires careful pointer management. The function maintains two pointers: curr (current node) and prev (previous node). When the target is found, there are three scenarios to handle: deleting the first node (prev is NULL, so inventory must be updated), deleting a middle node (prev->next bypasses curr), or deleting the last node (same bypass, but curr->next is NULL). In all cases, the freed memory is returned to the system.

```
void deleteProduct(char* id) {
    Product* curr = inventory;
    Product* prev = NULL;
    while (curr != NULL) {
        if (strcmp(curr->id, id) == 0) {
            if (prev == NULL)
                inventory = curr->next;  // Deleting first node
            else
                prev->next = curr->next;  // Bypass current node
            free(curr);  // Free memory
            printf("Product '%s' deleted!\n", id);
            return;
        }
        prev = curr;
        curr = curr->next;
    }
    printf("Product '%s' not found!\n", id);
}
```

Visual deletion scenarios:

```
Scenario 1: Deleting FIRST product (P001)

BEFORE:  inventory -> P001 -> P002 -> P003 -> NULL
AFTER:   inventory -> P002 -> P003 -> NULL
         (inventory now points to P002)

Scenario 2: Deleting MIDDLE product (P002)

BEFORE:  inventory -> P001 -> P002 -> P003 -> NULL
                           ^prev   ^curr
AFTER:   inventory -> P001 -> P003 -> NULL
         (prev->next now points to P003, bypassing P002)

Scenario 3: Deleting LAST product (P003)

BEFORE:  inventory -> P001 -> P002 -> P003 -> NULL
                                      ^prev  ^curr
AFTER:   inventory -> P001 -> P002 -> NULL
         (prev->next = curr->next which is NULL)
```

7.5 updateProduct() - Interactive Product Updates

The updateProduct function provides an interactive menu for modifying product details. It first locates the product using findProduct, then enters a loop presenting options to add stock, remove stock, update price, or cancel. Each modification immediately calls saveAllData() to persist changes. The remove stock option validates that the removal quantity does not exceed current stock, preventing negative inventory.

```
void updateProduct(char* id) {
    Product* p = findProduct(id);
    if (p == NULL) { printf("Product not found!\n"); return; }
    int choice;
    do {
        clearScreen();
        printf("Product: %s (ID: %s)\n", p->name, p->id);
        printf("Current Qty: %d  Price: $%.2f\n", p->quantity, p->price);
        printf("1. Add Stock\n2. Remove Stock\n3. Update Price\n4. Cancel\n");
        choice = readInt("Choice: ", 1, 4);
        switch(choice) {
            case 1:
                p->quantity += readInt("Qty to ADD: ", 0, 9999);
                saveAllData(); break;
            case 2:
                p->quantity -= readInt("Qty to REMOVE: ", 0, p->quantity);
                saveAllData(); break;
            case 3:
                p->price = readFloat("New price: $");
                saveAllData(); break;
        }
        if (choice != 4) wait();
    } while (choice != 4);
}
```

7.6 generateProductId() - Auto-generating IDs

The generateProductId function ensures every new product gets a unique ID by scanning all existing products to find the highest numbered ID, then incrementing by one. It extracts the numeric portion from each ID string (e.g., "P001" yields 1, "P045" yields 45), tracks the maximum, and formats the next number as a zero-padded four-digit string.

```
void generateProductId(char* id) {
    int maxNum = 0;
    Product* curr = inventory;
    while (curr != NULL) {
        int num = 0;
        for (int i = 1; i < strlen(curr->id); i++) {
            if (isdigit(curr->id[i]))
                num = num * 10 + (curr->id[i] - '0');
        }
        if (num > maxNum) maxNum = num;
        curr = curr->next;
    }
    nextProductNumber = maxNum + 1;
    sprintf(id, "%04d", nextProductNumber);
}
```

ID extraction examples:

```
ID: "P001"  -> skip 'P' -> 0*10+0=0, 0*10+0=0, 0*10+1=1 -> Result: 1
ID: "P045"  -> skip 'P' -> 0*10+0=0, 0*10+4=4, 4*10+5=45 -> Result: 45
ID: "P123"  -> skip 'P' -> 0*10+1=1, 1*10+2=12, 12*10+3=123 -> Result: 123

After finding maxNum = 45:
nextProductNumber = 46
sprintf(id, "%04d", 46) -> "0046"
Final ID: "P0046" (prefix added by caller)
```

Chapter 8: Shopping Cart System

8.1 The Cart Structure

The shopping cart is implemented as a linked list of OrderItem nodes. The global variable currentCart points to the head of this list. The cart exists only in memory during the current session and is cleared after an order is placed. This temporary nature means cart contents do not persist between program runs, which mirrors real-world shopping cart behavior on e-commerce websites.

```
currentCart (global variable)
    |
    v
+-------------------------------------------------------------+
| OrderItem #1                                                |
+-------------------------------------------------------------+
| productId: "P001"                                           |
| productName: "Gaming Laptop"                                |
| quantity: 2                                                 |
| price: 999.99                                               |
| next: -----------------+                                    |
+------------------------+------------------------------------+
                         |
                         v
+-------------------------------------------------------------+
| OrderItem #2                                                |
+-------------------------------------------------------------+
| productId: "P002"                                           |
| productName: "Wireless Mouse"                                |
| quantity: 1                                                 |
| price: 29.99                                                |
| next: NULL                                                  |
+-------------------------------------------------------------+
```

8.2 addToCart() - Adding Items

The addToCart function first validates that the product exists and has sufficient stock. It then checks if the product is already in the cart, in which case it increments the quantity rather than creating a duplicate entry. For new items, it creates a new OrderItem node and inserts it at the FRONT of the list (not the end) because front insertion is simpler and faster - it only requires updating the head pointer.

```
void addToCart(char* productId, int qty) {
    Product* p = findProduct(productId);
    if (p == NULL) { printf("Product not found!\n"); return; }
    if (p->quantity < qty) {
        printf("Not enough stock! Only %d available.\n", p->quantity);
        return;
    }
    // Check if already in cart
    OrderItem* curr = currentCart;
    while (curr != NULL) {
        if (strcmp(curr->productId, productId) == 0) {
            curr->quantity += qty;  // Update existing
            printf("Updated quantity in cart!\n");
            return;
        }
        curr = curr->next;
    }
    // Add new item at the beginning
    OrderItem* newItem = malloc(sizeof(OrderItem));
    strcpy(newItem->productId, p->id);
    strcpy(newItem->productName, p->name);
    newItem->quantity = qty;
    newItem->price = p->price;
    newItem->next = currentCart;
    currentCart = newItem;
    printf("Added '%s' to cart!\n", p->name);
}
```

Adding to cart visualization:

```
Empty cart:
currentCart ----> NULL

Add first item (Laptop):
currentCart ----> [Laptop] ----> NULL

Add second item (Mouse):
currentCart ----> [Mouse] ----> [Laptop] ----> NULL
  (Mouse inserted at front, faster!)

Add Laptop again (already in cart):
Find Laptop, increase quantity from 2 to 3
No new node created!
currentCart ----> [Mouse] ----> [Laptop (qty=3)] ----> NULL
```

8.3 showCart() - Display Cart

The showCart function displays the shopping cart contents in a formatted table, calculating subtotals for each item and a grand total. It traverses the cart linked list, multiplying each item price by quantity for the subtotal, and accumulating the total. The formatted output provides customers with a clear summary of their intended purchase.

```
void showCart() {
    if (currentCart == NULL) {
        printf("\nYour cart is empty!\n");
        return;
    }
    float total = 0;
    printf("\n+============================================+\n");
    printf("| # | ID       | Product            | Subtotal|\n");
    printf("+============================================+\n");
    OrderItem* curr = currentCart;
    int itemNum = 1;
    while (curr != NULL) {
        float sub = curr->price * curr->quantity;
        printf("| %d | %-8s | %-18s | %7.2f |\n",
               itemNum, curr->productId, curr->productName, sub);
        total += sub;
        curr = curr->next;
        itemNum++;
    }
    printf("+============================================+\n");
    printf("| TOTAL: $%30.2f |\n", total);
    printf("+============================================+\n");
}
```

Sample output:

```
+============================================================+
|                    YOUR SHOPPING CART                      |
+============================================================+
| # | ID         | Product              |   Qty |   Subtotal |
+------------------------------------------------------------+
| 1  | P002      | Wireless Mouse       |     1 |      29.99 |
| 2  | P001      | Gaming Laptop        |     2 |    1999.98 |
+------------------------------------------------------------+
|                                  TOTAL: $   2029.97 |
+============================================================+
```

8.4 removeCartItem() - Removing from Cart

The removeCartItem function removes an item by its position number (1-based index) in the cart display. It traverses the list while maintaining a counter, and when the target position is reached, it performs the standard linked list deletion using the prev/curr pointer pattern. The freed memory is returned to the system

immediately.

```
void removeCartItem(int itemNumber) {
    OrderItem* curr = currentCart;
    OrderItem* prev = NULL;
    int i = 1;
    while (curr != NULL) {
        if (i == itemNumber) {
            if (prev == NULL) currentCart = curr->next;
            else prev->next = curr->next;
            free(curr);
            printf("Item removed from cart!\n");
            return;
        }
        prev = curr;
        curr = curr->next;
        i++;
    }
    printf("Item not found!\n");
}
```

8.5 clearCart() - Empty Entire Cart

The clearCart function frees all nodes in the cart linked list by traversing from head to tail, saving each pointer in a temporary variable before advancing and freeing the previous node. After the loop, it sets currentCart to NULL. This function is called after a successful order placement to reset the cart for the next shopping session.

```
void clearCart() {
    OrderItem* curr = currentCart;
    while (curr != NULL) {
        OrderItem* temp = curr;
        curr = curr->next;
        free(temp);  // Free each node
    }
    currentCart = NULL;
}
```

Memory deallocation visualization:

```
Before:  currentCart -> [A] -> [B] -> [C] -> NULL

Step 1: temp=A, curr moves to B, free(A)
         currentCart -> [X] -> [B] -> [C] -> NULL
                        ^temp (freed)

Step 2: temp=B, curr moves to C, free(B)
         currentCart -> [X] -> [X] -> [C] -> NULL

Step 3: temp=C, curr moves to NULL, free(C)
         currentCart -> [X] -> [X] -> [X] -> NULL

After loop: currentCart = NULL (all memory freed)
```

Chapter 9: Order Management System

9.1 placeOrder() - Creating an Order

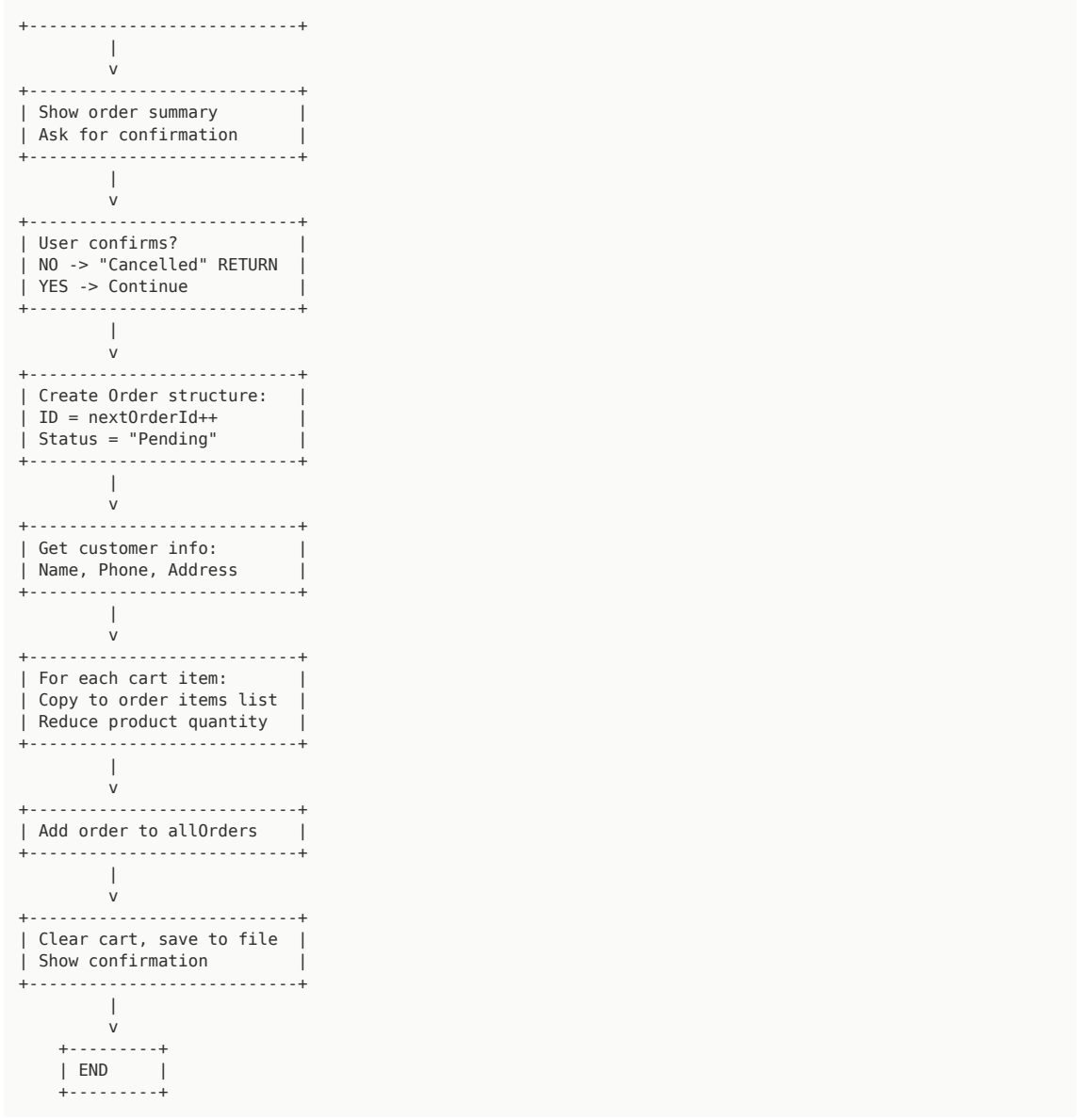
The placeOrder function is the most complex in the entire system, orchestrating nine distinct steps to transform a shopping cart into a confirmed order. It validates the cart is non-empty, displays an order summary, obtains customer confirmation, collects delivery information, copies cart items to the order,

reduces inventory stock, appends the order to the global orders list, clears the cart, and saves everything to disk. This function demonstrates how multiple data structures (cart list, order structure, inventory list) interact during a single business transaction.

```
void placeOrder() {
    // Step 1: Check cart
    if (currentCart == NULL) {
        printf("\nCart is empty!\n"); return;
    }
    // Step 2: Display summary
    float total = getCartTotal();
    printf("\nORDER SUMMARY\n");
    OrderItem* ci = currentCart;
    while (ci != NULL) {
        printf("  %s x%d = $%.2f\n", ci->productName, ci->quantity,
               ci->price * ci->quantity);
        ci = ci->next;
    }
    printf("  TOTAL: $%.2f\n", total);
    // Step 3: Confirm
    char confirm; scanf(" %c", &confirm); getchar();
    if (tolower(confirm) != 'y') { printf("Cancelled!\n"); return; }
    // Step 4: Create order
    Order* order = malloc(sizeof(Order));
    order->id = nextOrderId++;
    order->orderDate = time(NULL);
    order->total = total;
    strcpy(order->status, "Pending");
    order->items = NULL; order->next = NULL;
    // Step 5: Customer info
    readString("Name: ", order->customerName, MAX_NAME);
    readString("Phone: ", order->phone, MAX_PHONE);
    readString("Address: ", order->address, MAX_ADDRESS);
    // Step 6: Move items from cart to order, reduce stock
    ci = currentCart;
    while (ci != NULL) {
        OrderItem* item = malloc(sizeof(OrderItem));
        *item = *ci;  // Copy structure
        item->next = order->items;
        order->items = item;
        Product* p = findProduct(ci->productId);
        if (p != NULL) p->quantity -= ci->quantity;
        ci = ci->next;
    }
    // Step 7: Add to orders list
    if (allOrders == NULL) allOrders = order;
    else {
        Order* c = allOrders;
        while (c->next != NULL) c = c->next;
        c->next = order;
    }
    // Step 8-9: Clear cart, save, confirm
    clearCart(); saveAllData();
    printf("\nORDER #%d PLACED! Total: $%.2f\n", order->id, order->total);
}
```

Step-by-step flow diagram:

```
+-------------------------------------------------------------+
|                     PLACE ORDER FLOW                        |
+-------------------------------------------------------------+
    +---------+
    | START   |
    +----+----+
         |
         v
+--------------------------+
| Is cart empty?           |
| YES -> "Cart empty" RETURN|
| NO  -> Continue          |
```



9.2 findOrder() - Searching Orders

The findOrder function searches the allOrders linked list for an order with the specified ID. Unlike findProduct which uses string comparison, this function compares integer IDs directly using the == operator, making it slightly simpler and faster.

```
Order* findOrder(int id) {
    Order* curr = allOrders;
    while (curr != NULL) {
        if (curr->id == id) return curr;
        curr = curr->next;
    }
    return NULL;
}
```

9.3 cancelOrder() - Cancelling an Order

The cancelOrder function implements important business logic. Before cancelling, it checks the order status - orders that are already "Shipped" or "Delivered" cannot be cancelled. This prevents logical inconsistencies like trying to cancel something that is already in transit. Crucially, when an order is cancelled, the function restores inventory by adding the ordered quantities back to each product. This ensures stock accuracy is maintained even when orders are reversed.

```
void cancelOrder(int id) {
    Order* order = findOrder(id);
    if (order == NULL) { printf("Order #%d not found!\n", id); return; }
    if (strcmp(order->status, "Delivered") == 0 ||
        strcmp(order->status, "Shipped") == 0 ||
        strcmp(order->status, "Cancelled") == 0) {
        printf("Cannot cancel order in current status!\n");
        return;
    }
    // Restore inventory
    OrderItem* item = order->items;
    while (item != NULL) {
        Product* p = findProduct(item->productId);
        if (p != NULL) p->quantity += item->quantity;
        item = item->next;
    }
    strcpy(order->status, "Cancelled");
    saveAllData();
    printf("Order #%d cancelled! Inventory restored.\n", id);
}
```

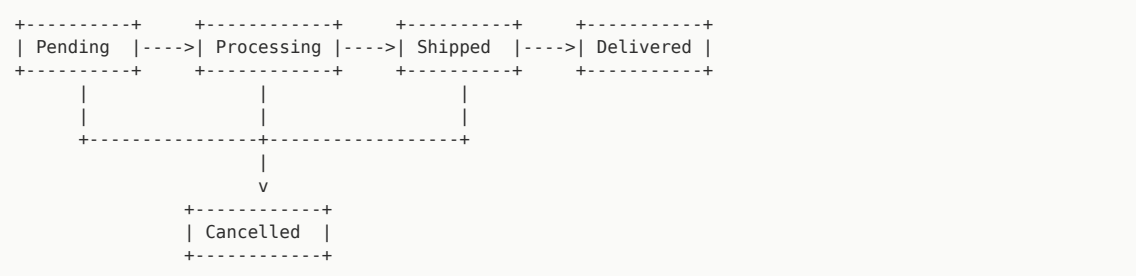
> Important: Without inventory restoration, cancelling orders would cause permanent stock loss. If a customer buys 2 laptops (stock goes from 50 to 48) then cancels, the stock must go back to 50.

9.4 updateOrderStatus() - Changing Status

The updateOrderStatus function changes an order status string to a new value. Orders follow a lifecycle from "Pending" through "Processing", "Shipped", and finally "Delivered". At any point before shipping, an order can also transition to "Cancelled".

```
void updateOrderStatus(int id, char* newStatus) {
    Order* order = findOrder(id);
    if (order == NULL) { printf("Order #%d not found!\n", id); return; }
    strcpy(order->status, newStatus);
    saveAllData();
    printf("Order #%d status -> '%s'\n", id, newStatus);
}
```

Order status flow:



9.5 showOrderDetails() - Display Order

The showOrderDetails function displays complete order information including customer details, order status, date, all items with subtotals, and the grand total. The ctime() function converts the Unix timestamp stored in orderDate to a human-readable date string.

```
void showOrderDetails(Order* order) {
    printf("\nORDER DETAILS #%d\n", order->id);
    printf("Customer: %s\n", order->customerName);
    printf("Phone: %s\n", order->phone);
    printf("Address: %s\n", order->address);
    printf("Status: %s\n", order->status);
    printf("Date: %s", ctime(&order->orderDate));
    printf("ITEMS:\n");
    OrderItem* item = order->items;
    int n = 1;
    while (item != NULL) {
        printf("  %d. %s x%d = $%.2f\n", n, item->productName,
               item->quantity, item->price * item->quantity);
        item = item->next; n++;
    }
    printf("TOTAL: $%.2f\n", order->total);
}
```

Chapter 10: Customer Portal - The User Interface

10.1 customerMenu() - Main Customer Menu

The customerMenu function is the entry point for customer interactions. It displays a menu with four options: browse and buy products, view cart and checkout, view order history, or return to the main menu. The menu uses a do-while loop to keep displaying until the customer chooses to go back, with clearScreen() calls between each action to maintain a clean interface.

```
void customerMenu() {
    int choice;
    do {
        clearScreen();
        printf("+====================================+\n");
        printf("|       CUSTOMER PORTAL              |\n");
        printf("+====================================+\n");
        printf("1. Browse & Buy Products\n");
        printf("2. View Cart & Checkout\n");
        printf("3. My Orders\n");
        printf("4. Back to Main Menu\n");
        choice = readInt("\nChoice: ", 1, 4);
        switch (choice) {
            case 1: browseAndBuy(); break;
            case 2: viewCartAndCheckout(); break;
            case 3: viewMyOrders(); break;
        }
    } while (choice != 4);
}
```

10.2 browseAndBuy() - Shopping Experience

The browseAndBuy function provides the core shopping experience. Customers can choose to view all products or search by name/category. After viewing products, they can enter product IDs and quantities to add items to their cart. The function validates product IDs using verifyProduct() and loops to allow adding multiple products before returning to the browse menu.

```
void browseAndBuy() {
    int choice;
    char searchTerm[MAX_NAME];
    do {
        clearScreen();
        printf("1. Show All Products\n2. Search\n3. Back\n");
        choice = readInt("\nChoice: ", 1, 3);
        if (choice == 1) {
            showProducts();
            char addMore = 'y';
            while (addMore == 'y' || addMore == 'Y') {
                char id[MAX_ID]; int qty;
                while (1) {
                    printf("\nProduct ID (0 to exit): ");
                    scanf("%s", id); getchar();
                    if (strcmp(id, "0") == 0) return;
                    if (verifyProduct(id)) break;
                    printf("Product '%s' not found!\n", id);
                }
                qty = readInt("Quantity: ", 1, 999);
                addToCart(id, qty);
                printf("\nAdd another? (y/n): ");
                scanf(" %c", &addMore); getchar();
            }
            wait();
        }
        else if (choice == 2) {
            readString("Search term: ", searchTerm, MAX_NAME);
            searchAndShowProducts(searchTerm);
        }
    } while (choice != 3);
}
```

10.3 viewCartAndCheckout() - Cart Management

This function bridges the shopping experience and the checkout process. It displays the cart contents, then offers options to update item quantities, remove items, proceed to checkout (which calls placeOrder()), or go back. The function handles the edge case where removing all items from the cart automatically returns to the previous menu.

```
void viewCartAndCheckout() {
    if (currentCart == NULL) {
        printf("\nYour cart is empty!\n"); wait(); return;
    }
    int choice;
    do {
        showCart();
        if (getCartItemCount() == 0) break;
        printf("\n1. Update quantity\n2. Remove item\n3. Checkout\n4. Back\n");
        choice = readInt("Choice: ", 1, 4);
        switch (choice) {
            case 1: {
                int num = readInt("Item # to update: ", 1, getCartItemCount());
                int qty = readInt("New quantity: ", 1, 999);
                updateCartItem(num, qty); break;
            }
            case 2: {
                int num = readInt("Item # to remove: ", 1, getCartItemCount());
                removeCartItem(num); break;
            }
            case 3: placeOrder(); wait(); return;
        }
        if (choice != 4) wait();
    } while (choice != 4);
}
```

10.4 viewMyOrders() - Order History

The viewMyOrders function displays all orders and allows the customer to view details of any specific order by entering its ID. From the detail view, the customer can also cancel an order if it is still in a cancellable status (Pending or Processing). This function provides customers with visibility into their purchase history and the ability to self-serve order cancellations.

```
void viewMyOrders() {
    if (allOrders == NULL) {
        printf("\nNo orders yet!\n"); wait(); return;
    }
    showAllOrders();
    int orderId = readInt("\nOrder ID to view (0 skip): ", 0, 9999);
    if (orderId > 0) {
        Order* order = findOrder(orderId);
        if (order != NULL) {
            showOrderDetails(order);
            printf("\n1. Cancel this order\n2. Back\n");
            int choice = readInt("Choice: ", 1, 2);
            if (choice == 1) cancelOrder(orderId);
        } else printf("Order #%d not found!\n", orderId);
    }
    wait();
}
```

Chapter 11: Admin Portal - The Manager Interface

11.1 adminMenu() - Main Admin Menu

The adminMenu function provides the administrative interface after successful OTP verification. It offers options for product management (add, view, update/delete) and order management (create, view, update status, cancel). The admin portal has broader permissions than the customer portal, allowing full CRUD operations on both products and orders.

```
void adminMenu() {
    int choice;
    do {
        clearScreen();
        printf("+====================================+\n");
        printf("|         ADMIN PORTAL               |\n");
        printf("+====================================+\n");
        printf("=== PRODUCT MANAGEMENT ===\n");
        printf("1. Add Product\n2. View All Products\n");
        printf("3. Product Operations\n");
        printf("\n=== ORDER MANAGEMENT ===\n");
        printf("4. Create Order\n5. View All Orders\n");
        printf("6. Order Operations\n\n7. Logout\n");
        choice = readInt("\nChoice: ", 1, 7);
        switch (choice) {
            case 1: adminAddProduct(); wait(); break;
            case 2: adminViewProducts(); break;
            case 3: /* Product submenu */ break;
            case 4: adminCreateorder(); wait(); break;
            case 5: adminViewOrders(); wait(); break;
            case 6: /* Order submenu */ break;
        }
    } while (choice != 7);
}
```

11.2 adminAddProduct() - Adding with Duplicate Check

The adminAddProduct function includes important duplicate detection logic. Before adding a new product, it searches the inventory for an existing product with the same name. If a duplicate is found, the admin is given the option to update the existing product instead, add it as a new product anyway, or cancel. This prevents data inconsistencies from duplicate entries while still allowing legitimate cases where products might share similar names.

```
void adminAddProduct() {
    char name[MAX_NAME];
    printf("\n=== ADD NEW PRODUCT ===\n");
    readString("Product Name (0 to cancel): ", name, MAX_NAME);
    if (strcmp(name, "0") == 0) return;

    Product* existing = findProductByName(name);
    if (existing != NULL) {
        printf("WARNING: '%s' already exists!\n", name);
        printf("1. Update existing\n2. Add as new\n3. Cancel\n");
        int choice = readInt("Choice: ", 1, 3);
        if (choice == 1) { updateProduct(existing->id); return; }
        else if (choice == 3) return;
    }

    Product* p = malloc(sizeof(Product));
    generateProductId(p->id);
    strcpy(p->name, name);
    readString("Category: ", p->category, MAX_CATEGORY);
    p->quantity = readInt("Quantity: ", 0, 99999);
    p->price = readFloat("Price: $");
    p->next = NULL;
    addProduct(p);
    saveAllData();
    printf("Product '%s' added!\n", name);
}
```

11.3 adminCreateorder() - Admin Creating Order

The adminCreateorder function allows admins to create orders on behalf of customers. This is useful for phone orders, in-store purchases, or any situation where the customer is not using the system directly. The admin selects products and quantities (using the same cart system), then provides customer delivery information to complete the order.

```
void adminCreateorder() {
    printf("\n=== ADMIN CREATE ORDER ===\n");
    showProducts();
    char addMore = 'y';
    while (addMore == 'y' || addMore == 'Y') {
        char id[MAX_ID]; int qty;
        printf("\nProduct ID to add: ");
        scanf("%s", id); getchar();
        if (!verifyProduct(id)) { printf("Not found!\n"); continue; }
        qty = readInt("Quantity: ", 1, 999);
        addToCart(id, qty);
        printf("\nAdd another? (y/n): ");
        scanf(" %c", &addMore); getchar();
    }
    if (currentCart == NULL) { printf("Cart empty.\n"); return; }
    // Create order with admin-provided customer info
    Order* order = malloc(sizeof(Order));
    order->id = nextOrderId++;
    order->orderDate = time(NULL);
    strcpy(order->status, "Pending");
    // ... collect customer info and build order ...
```

```
    clearCart(); saveAllData();
    printf("ORDER #%d CREATED!\n", order->id);
}
```

Chapter 12: The Main Function - Where It All Begins

The main function is the entry point of the entire program and the orchestrator of all system activity. It loads all persisted data when the program starts, then enters an infinite loop displaying the main menu. Based on the user choice, it routes to either the customer portal (no authentication required) or the admin portal (OTP verification required). When the user chooses to exit, it saves all data to ensure nothing is lost between sessions.

```
int main() {
    loadAllData();  // Load products and orders from files
    int choice;
    do {
        clearScreen();
        printf("+====================================+\n");
        printf("|   INVENTORY MANAGEMENT SYSTEM      |\n");
        printf("+====================================+\n");
        printf("\n1. Customer Portal\n");
        printf("2. Admin Portal\n");
        printf("3. Exit\n");
        choice = readInt("\nEnter your choice: ", 1, 3);
        switch (choice) {
            case 1: customerMenu(); break;
            case 2:
                if (verifyAdmin() == 1) adminMenu();
                else { printf("\nAccess Denied!\n"); getchar(); }
                break;
            case 3: saveAllData();
                     printf("\nThank you for using the system!\n");
                     break;
        }
    } while (choice != 3);
    return 0;
}
```

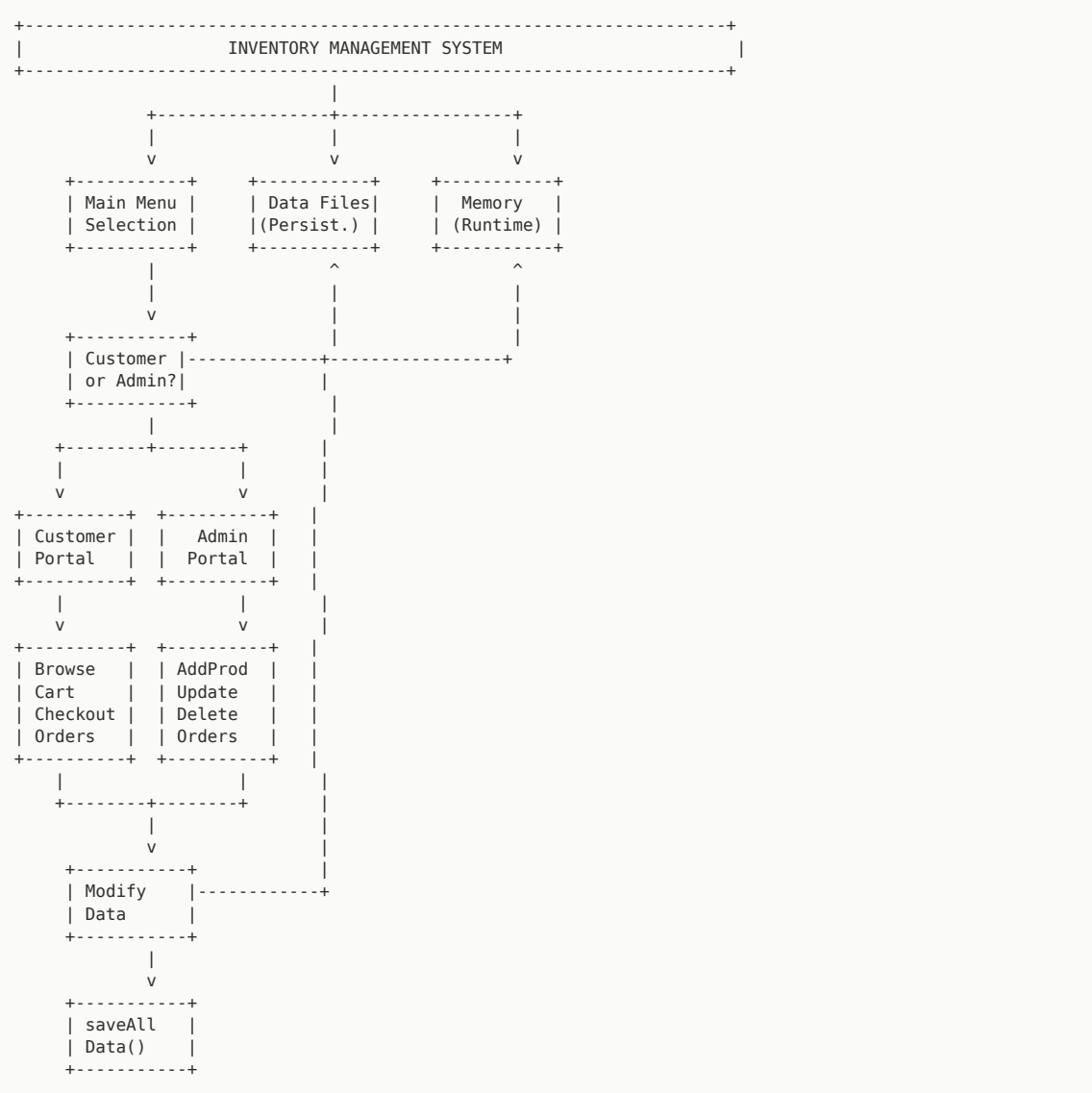
Program Execution Flow:

```
START
  |
  v
+-------------------------------------------------------------+
| main()                                                      |
| +-- loadAllData() --> Read products and orders from files    |
| +-- Enter main loop                                         |
+-------------------------------------------------------------+
  |
  v
+-------------------------------------------------------------+
| Show Main Menu:                                             |
| 1. Customer Portal                                          |
| 2. Admin Portal                                             |
| 3. Exit                                                     |
+-------------------------------------------------------------+
  |
  +--- Choice 1 ---> customerMenu() ---> Browse, Cart, Orders
  |
  +--- Choice 2 ---> verifyAdmin() ---> If success -> adminMenu()
  |                                    Products, Orders mgmt
  |
  +--- Choice 3 ---> saveAllData() ---> Exit
```

Chapter 13: Complete Flow Diagrams

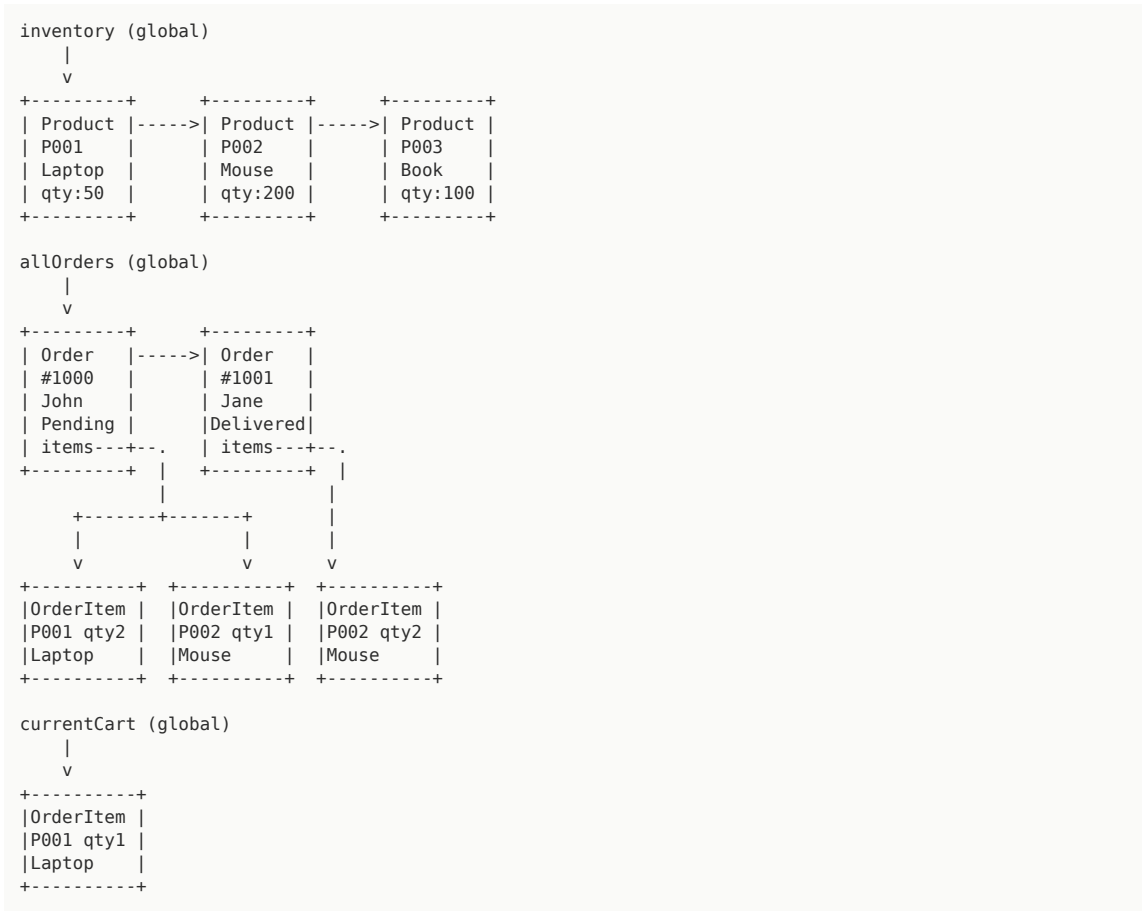
13.1 Complete System Architecture

The following diagram shows the complete architecture of the Inventory Management System. The main menu acts as the central hub, routing users to either the Customer Portal or the Admin Portal (after authentication). All modifications flow back to the data persistence layer through saveAllData(). This architecture demonstrates the separation of concerns principle, with distinct modules for customer operations, administrative operations, and data management.



13.2 Data Structure Relationships

This diagram illustrates how the four core data structures relate to each other through pointers. The inventory is a singly linked list of Products. allOrders is a singly linked list of Orders, where each Order contains a nested singly linked list of OrderItems. The currentCart is also a singly linked list of OrderItems, separate from order item lists. Understanding these relationships is essential for tracing data flow through the system.



13.3 Function Call Hierarchy

The function call hierarchy shows the complete calling tree from main() downward. Each indentation level represents a function call from the function above it. This tree is essential for understanding program flow during execution and for debugging. When an error occurs, following this hierarchy from the top helps identify which function in the call chain is responsible.

```
main()
|
+-- loadAllData()
|   +-- loadAdminData()
|   +-- [Read products and orders from file]
|
+-- customerMenu()
|   +-- browseAndBuy()
|   |   +-- showProducts()
|   |   +-- verifyProduct()
|   |   +-- addToCart()
|   |   |   +-- findProduct()
|   |   +-- searchAndShowProducts()
|   +-- viewCartAndCheckout()
|   |   +-- showCart()
|   |   +-- updateCartItem()
|   |   +-- removeCartItem()
|   |   +-- placeOrder()
|   |       +-- getCartTotal()
|   |       +-- clearCart()
|   |       +-- saveAllData()
|   +-- viewMyOrders()
|       +-- showAllOrders()
|       +-- findOrder()
|       +-- showOrderDetails()
|       +-- cancelOrder()
```

```
|
+-- adminMenu()
|   +-- adminAddProduct()
|   |   +-- generateProductId()
|   |   +-- addProduct()
|   |   +-- saveAllData()
|   +-- adminViewProducts()
|   |   +-- showProducts()
|   +-- adminUpdateProduct()
|   |   +-- findProduct()
|   |   +-- updateProduct()
|   |   +-- saveAllData()
|   +-- adminDeleteProduct()
|   |   +-- deleteProduct()
|   |   +-- saveAllData()
|   +-- adminViewOrders()
|   |   +-- showAllOrders()
|   |   +-- showOrderDetails()
|   +-- adminUpdateOrderStatus()
|   |   +-- updateOrderStatus()
|   |   +-- saveAllData()
|
+-- saveAllData()
    +-- saveAdminData()
    +-- [Write products and orders to file]
```

Chapter 14: Common Questions & Answers

This chapter addresses the most frequently asked questions about the Inventory Management System code. These questions cover fundamental C programming concepts, data structure decisions, and design patterns used throughout the project. Each answer explains not just the "what" but the "why" behind each design choice.

Q1: Why use linked lists instead of arrays?

A: Arrays have fixed size. If you declare Product products[100], you can only store 100 products. With linked lists, you can store unlimited products (until memory runs out). Linked lists also make insertion and deletion at any position efficient - just adjust a few pointers instead of shifting hundreds of elements. In a real store, products are constantly being added and removed, making linked lists the natural choice.

```
Product products[100];  // LIMITED to 100!
Product* inventory = NULL;  // UNLIMITED!
```

Q2: What happens to memory when we add a product?

A: malloc() allocates memory on the heap. sizeof(Product) is approximately 146 bytes (50+50+4+4+30+8). malloc finds 146 free bytes in heap memory and returns a pointer to that memory. This memory persists until explicitly freed with free(). Unlike stack variables that are automatically cleaned up when a function returns, heap memory must be managed manually by the programmer.

```
Product* p = malloc(sizeof(Product));  // ~146 bytes allocated
```

Q3: Why do we need to call free()?

A: Without free(), memory stays allocated even though you are not using it. Over time, the program would consume more and more memory (a memory leak). In this system, free() is called when deleting products (deleteProduct), removing cart items (removeCartItem), and clearing the cart (clearCart). Every malloc should eventually have a matching free.

Q4: Why does the cart use OrderItem instead of Product?

A: Because when you buy a product, we need to save the price AT PURCHASE TIME, not the current price, and the name AT PURCHASE TIME, not the current name. If we used a Product pointer, and later changed the product name or price, the order would show the wrong information. By copying data into OrderItem, the order becomes a snapshot frozen in time.

Q5: What does strtok() do in file loading?

A: strtok() splits a string into tokens using a delimiter character. It is used extensively in loadAllData() to parse pipe-delimited lines. The first call strtok(line, "|") gets the first token. Subsequent calls with NULL as the first argument continue from where the previous call left off.

```
char line[] = "P001|Gaming Laptop|Electronics|50|999.99";
// strtok(line, "|") -> "P001", NULL, "|" -> "Gaming Laptop", etc.
```

Q6: Why is there a next pointer in every structure?

A: That is what makes it a linked list! Each node points to the next node, creating a chain: Node1 -> Node2 -> Node3 -> NULL. Without the next pointer, you could not traverse from one node to another. The next pointer is the fundamental mechanism that enables dynamic data storage in a linked list.

Q7: What is the difference between readInt() and scanf()?

A: scanf() is dangerous because if the user types "abc" instead of a number, the variable is left uninitialized, potentially causing crashes. readInt() reads input as a string first, validates every character is a digit, checks the range, and only then converts to an integer. If anything is wrong, it shows an error and asks again, making the program virtually crash-proof.

```
scanf("%d". &x):  // If "abc" entered -> x is UNINITIALIZED!
readInt("Enter: ", 1, 10);  // Validates, handles errors, loops
```

Q8: Why does placeOrder() copy items instead of moving pointers?

A: Because the cart is cleared after an order (using clearCart which calls free()). If we just moved pointers, the order items would point to the same memory as the cart items, and clearing the cart would free that memory. The order would then have dangling pointers pointing to freed memory, causing crashes or data corruption when accessed later.

Q9: What is time_t and ctime()?

A: time_t stores seconds since January 1, 1970 (Unix timestamp). time(NULL) returns the current timestamp as an integer. ctime() converts this integer into a human-readable string like "Fri Oct 25 14:30:00 2024". This system is used to record when orders were placed.

```
time t now = time(NULL);     // e.g., 1729876543
printf("%s", ctime(&now));   // "Fri Oct 25 14:30:00 2024"
```

Q10: Why do we need saveAllData() and loadAllData()?

A: Memory (RAM) is volatile - when the program exits, all data is lost. saveAllData() writes the current state to disk (permanent storage) so data survives between sessions. loadAllData() reads from disk when the program starts. Without these functions, every time you run the program, you would start with empty inventory and no order history.

Q11: What does strcspn() do in readString()?

A: strcspn() finds the position of the first occurrence of any character from a set within a string. In readString(), it finds the position of the newline character (\n) that fgets() captures, so it can be replaced with a null terminator (\0). This effectively strips the trailing newline from user input.

```
char input[] = "John Doe\n";
// strcspn(input, "\n") returns 8; input[8] = '\0'; -> "John Doe\0"
```

Q12: How does OTP verification work?

A: The system generates a random 6-digit number using srand(time(NULL)) and rand(). It displays this number on screen (simulating an SMS). The user then enters the OTP, and the system compares the entered value with the generated value. If they match, access is granted. In a real system, the OTP would be sent via SMS gateway, would expire after a timeout, and would be rate-limited.

Q13: What happens if two admins try to login simultaneously?

A: This is a single-user, single-threaded program with no concurrency handling. Only one person can use the system at a time. For multi-user support, you would need file locking (to prevent data corruption), a database system (instead of text files), and a client-server architecture with proper session management.

Q14: Why does nextOrderId start at 1000?

A: Starting at 1000 makes order IDs look professional and readable. Order numbers like 1000, 1001, 1002 are easier to communicate and remember than 1, 2, 3. It also leaves room for future expansion (e.g., if you later want to add a different order type with IDs starting from 1).

Q15: How does generateProductId() ensure unique IDs?

A: It scans ALL existing products, extracts the numeric portion from each ID, finds the highest number, adds 1, and formats it as a 4-digit string. Since it considers every existing ID, it can never generate a duplicate. For example, if existing IDs are P001, P002, P045, the max is 45, so the next ID will be P0046.

```
// If existing IDs: P001, P002, P045
// maxNum = 45, sprintf(id, "%04d", 46) -> "0046" -> Final: "P0046"
```

Appendix: Quick Reference

Color Codes Used

The system uses ANSI escape codes to add colors to the terminal output. These codes work on most modern terminals and provide visual feedback to the user. The #define macros make the code more readable than using raw escape sequences.

```
#define RED     "\033[1;31m"
#define GREEN   "\033[1;32m"
#define YELLOW  "\033[1;33m"
#define BLUE    "\033[1;34m"
#define MAGENTA "\033[1;35m"
#define CYAN    "\033[1;36m"
#define RESET   "\033[0m"
```

Common Function Patterns

These five patterns appear repeatedly throughout the codebase and are fundamental to understanding how linked list operations work in C:

Pattern	Code Example
Traverse list	while (curr != NULL) { ... curr = curr->next; }
Add to front	new->next = head; head = new;
Add to back	while (curr->next) curr = curr->next; curr->next = new;
Find in list	if (strcmp(curr->id, id) == 0) return curr;
Delete from list	prev->next = curr->next; free(curr);

Table A.1: Common linked list operation patterns

File Format Summary

The system uses two text files for data persistence, both using pipe-delimited format with section markers:

```
store_data.txt:
========================================
#PRODUCTS
ID|Name|Category|Quantity|Price
#END

#ORDERS
ID|Customer|Phone|Address|Total|Status|Timestamp
#END

admin_data.txt:
========================================
OutletID|Phone
```